Distributed NoSQL Triple Store Prototype: Architectural Design and Implementation

charan sri sai(IMT2021002)
Akhil(IMT2021021)
Mayank(IMT2021073)
chokshi(IMT2021012)

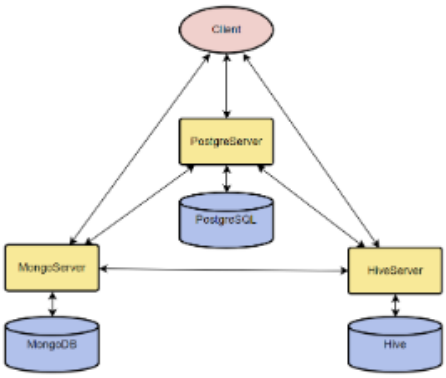
I. Introduction

The problem statement entails the design and implementation of a distributed NoSQL triple store prototype, emphasizing the utilization of state-based objects. This endeavor addresses the growing need for efficient management and querying of data organized in triple form, comprising subject, predicate, and object components. The significance of this project lies in its potential to offer scalable and robust solutions for handling diverse datasets in distributed environments, catering to the increasing demand for flexible data storage and retrieval mechanisms.

The project's primary objectives include developing a prototype capable of performing essential operations such as querying, updating, and merging triples across multiple servers,and enabling synchronization between servers via a merge function, each employing distinct frameworks for data processing and storage.This project also aims to construct a Triple datastore capable of storing subject, predicate, object triplets across multiple server locations. Replication ensures the availability of the datastore even if one server goes offline.

II. System Architecture

We have designed a distributed nosql system with three servers, all servers containing the same data. Each server has a database with two tables one to store actual data having columns subject,predicate and object, and the other table is to store information about the updates done by clients.This updates table is used in merge and update operations.



Client can communicate with the server of it's choice and perform operations like querying,updating the server,merging two servers. We have implemented socket programming to ease the process of communication between client and server. client can close connection with the current sever it's communicating at any time and start comminicating with the other server.

Server offers services like querying rows having particular subject,updating object value of rows having given subject and preidcate values and merging the current server with other server i.e sychronizing two servers.

We have used state-based objects as data to be stored in the servers.Each server contains data of format $< subject > < predicate > < object >$. State-based objects play a crucial role in maintaining consistency and integrity within the system. Each server maintains its own state-based object, representing the current state of the triple store on that server. These state-based objects are updated whenever merge is called, ensuring that all servers remain synchronised and consistent.

III. Methodology

We have implemented the methods query,update and merge for state based objects. Whenever a client wants to perform query operation with the server it's connected with, it provides the server $< subject >$ value,then server returns the client with all rows having the given subject value.

Similarly in update operation client provides the server with $< subject >$, $< predicate >$ and $< object >$ values, then server performs update operation by changing the object value in the rows having same subject and predicate values in the data table and stores this update information done by the client in updates table. updates table contains columns timestamp,subject,predicate and object. If updates log table already contains a row with the same subject and predicate, it updates timestamp and object values.+

In the merge operation, let's say client is in connection with the server 2, and wants to perform merge operation. Now, server 2 asks the client with which server should the merge operation be performed. Let's say client replies saying merge operation should be performmed with the server 3.Now our task in merge operation is to synchronize server 2 with server 3,which means update server 2 with changes made by server 3,if the updates in server 3 are most recent compared to server 2.Now, server 2 gets the updates information table from server

3. server 2 iterates through the updates log table of server 3 and checks if that row is present in it's updates log table. If yes, then it compares timestamps of both the rows. If timestamp of server 3 is greater than server 2, performs update information, else doesn't perform update operation. If corresponding row is not present in the updates log of server 2, it indicates, that row hasn't been updated yet. Then also it performs the update operation.

We have implemented hive,mongo and postgresql databases for three servers, one for each server. We used python to connect with the databases in servers and perform operations.

Reasons for Choosing Databases

Hive:

- Scalability: Hive is built on top of Hadoop, providing scalability by distributing data across multiple nodes in a cluster.
- Integration with Hadoop Ecosystem: Hive seamlessly integrates with Hadoop components such as HDFS and YARN, offering a cohesive big data processing environment.

MongoDB:

- Flexible Schema Design: MongoDB's document-oriented structure allows for flexible schema design, suitable for storing triples with varying structures.
- High Performance: MongoDB is optimized for high performance with features like indexing and sharding, enhancing query performance and scalability.
- Horizontal Scalability: MongoDB supports horizontal scalability through sharding, facilitating the distribution of data across multiple nodes.

PostgreSQL:

- ACID Compliance: PostgreSQL ensures data consistency, integrity, and durability, essential for a triple store.
- Rich Feature Set: PostgreSQL offers advanced features such as complex queries, indexing, and support for advanced data types.
- Mature Ecosystem: PostgreSQL has a mature ecosystem with extensive community support, documentation, and third-party tools.

Choosing n=3, meaning three servers, for our distributed NoSQL triple store project offers several key advantages. Firstly, it simplifies the management of our system, making it easier to set up, maintain, and troubleshoot. This simplicity is crucial for us as students, as it reduces the learning curve and allows us to focus more on understanding the fundamental concepts of distributed systems. With fewer servers, we can also better manage our resources, especially if we're operating within certain constraints such as budget or hardware availability. Additionally, working with a smaller cluster size enables us to concentrate our efforts on optimizing performance, scalability, and fault tolerance without being overwhelmed by complexity. Moreover, starting with n=3 provides us with a solid foundation for scalability preparation, allowing us to design our system with future growth in mind. Overall, choosing n=3 strikes a balance between simplicity, educational value, resource constraints, and scalability preparation, making it a pragmatic choice for our project

We have chosen python for connecting with databases due to its simplicity, extensive library support, cross-platform compatibility, flexibility, and strong community. Its clear and concise syntax makes it easy to write and maintain database connection code. With a rich ecosystem of database libraries like SQLAlchemy, psycopg2, pymongo, and pyhive,it provides high-level abstractions and convenient APIs for interacting with various database systems, reducing boilerplate code and accelerating development.

Programming Language and Frameworks:

1) **Python:** Python is chosen for its versatility, ease of use, and extensive library support. It offers libraries for socket programming, database interaction.
2) **Socket Programming:** Python's built-in `socket` library is utilized for establishing communication channels between the client and server, as well as among servers. Its simplicity and reliability make it well-suited for implementing TCP connections, ensuring secure data transmission.
3) **Database Libraries:** Different databases require different libraries for interaction. Python libraries such as `pymongo` for MongoDB, `psycopg2` for PostgreSQL, `pyhive` for interacting with Apache Hive are selected based on their compatibility and functionality with each respective database.

IV. Implementation

The servers interact with their respective databases using Python libraries tailored to each specific database management system. These libraries enable seamless communication between the servers and their associated databases, facilitating operations such as data retrieval, and modification.

The communication channels between the client and server, as well as among the servers themselves, are established using socket programming with TCP connections. TCP (Transmission Control Protocol) connections are employed to ensure reliable and orderly data transmission, minimising the risk of data loss or corruption during transfer. This approach guarantees robust and secure communication between the various components of the system, facilitating seamless interaction and data exchange.

Each server follows a modular implementation, ensuring consistency in how they handle client requests. The key difference lies in how each server interacts with its respective database, which is facilitated by the use of different libraries tailored to the specific database management system (DBMS) employed by that server. Despite these variations in database interaction, the overall behaviour of each server in response to client queries and updates remains similar due to the modular design of the server implementation. Below is an overview of the implementation of different functionalities for the client-server interaction:

Query

The client sends a JSON object containing the subject being queried and client-id to the server. The server reads the query request and queries the respective database for the subject-predicate-object triplet. It then sends back the query result to the client as a JSON object in response.

Update

The client sends a JSON object containing the subject-predicate pair and object being updated, timestamp of the request, and client-id to the server. When the server receives an update request,first it deletes the rows having given subject and predicate.Then, it inserts new row with given subject,predicate and object values. And, then it checks if there is a row with same subject and predicate in the update log table,if yes, updates the timestamp and object values else adds new update log. It then sends an acknowledgment response back to the client.

Merge

The client sends a JSON object containing the server-id to merge with, timestamp of the request, and client-id to the server. The server connected to the client (hereafter referred to as server1) reads the merge request and gets the update log table of the server with which client wants to merge(server2). Now server1 iterates through the updates log of server2 and checks if there is a row with same subject and predicate in updates log of server1,if yes compares timestamps of both, if timestamp of server2 is greater than server1 performs the update operation as described above, else continues. if there is no row with same subject and predicate in the updates log of server1 , then also performs the update operation. It then sends an acknowledgement response back to the client.

Below are the code snippets for query,update and merge operations:

```
def query(self, subject):
    print('Querying Hive server')
    query = f"SELECT * FROM {self.table_name} WHERE subject = '{subject}'"
    self.cursor.execute(query)
    results_list = [dict(zip([column[0] for column in self.cursor.description], row)) for row in self.cursor.fetchall()]
    return results_list
```

```
def update(self, subject, predicate, obj,timestamp):
    print('Updating Hive server')
    #update_query =f"INSERT OVERWRITE TABLE {self.table_name} SELECT subject, predicate, CASE WHEN subject = '{subject}' AND predicate = '{predicate}
    delete_query = f"""
        INSERT OVERWRITE TABLE {self.table_name}
        SELECT * FROM {self.table_name}
        WHERE subject != '{subject}' OR predicate != '{predicate}'
        """
    self.cursor.execute(delete_query)
    update_query=f"INSERT INTO {self.table_name} (subject, predicate, object) VALUES ('{subject}', '{predicate}', '{obj}')"
    self.cursor.execute(update_query)
    self.conn.commit()
    print('Entry updated')
    self.cursor.execute(f"SELECT * FROM updates_table WHERE subject='{subject}' AND predicate='{predicate}'")
    existing_row = self.cursor.fetchone()

    if existing_row:
        # Update existing row using INSERT OVERWRITE
        delete_query = f"""
        INSERT OVERWRITE TABLE updates_table
        SELECT * FROM updates_table
        WHERE subject != '{subject}' OR predicate != '{predicate}'
        """
        self.cursor.execute(delete_query)

    # Insert new row with current timestamp
    #timestamp = datetime.now().strftime('%Y-%m-%d-%H-%M-%S')
    insert_query = f"INSERT INTO updates_table (timestp, subject, predicate, object) VALUES ('{timestamp}', '{subject}', '{predicate}', '{obj}')"
    self.cursor.execute(insert_query)

    self.conn.commit()
    print('Entry updated or inserted')
```

```
def merge(self, server_id):
    if server_id == 1:
        mongo_client = pymongo.MongoClient('localhost', 27017)
        mongo_db = mongo_client['sample_db']
        mongo_collection = mongo_db['updates']
        mongo_data = mongo_collection.find().sort('timestp', pymongo.DESCENDING)
        hive_conn = hive.Connection(host='localhost', port=10000, database='yago_db')
        hive_cursor = hive_conn.cursor()

        for doc in mongo_data:
            subject = doc['subject']
            predicate = doc['predicate']
            obj = doc['object']
            timestp = doc['timestp']
            hive_cursor.execute(f"SELECT * FROM updates_table WHERE subject='{subject}' AND predicate='{predicate}'")
            hive_row = hive_cursor.fetchone()

            if (hive_row and timestp > hive_row[0]):
                self.update(subject,predicate,obj,timestp)
            elif hive_row is None:
                self.update(subject,predicate,obj,timestp)

        hive_cursor.close()

        mongo_client.close()
```

Data Structures used to store the triples:

- **PostgreSQL server:** Stores triples in a table with 3 columns, where each row is assigned for each triple. The table structure is straightforward, with columns representing the subject, predicate, and object components of the triple.
- **MongoDB server:** Stores data in collections, with each collection containing JSON structured documents representing individual triples. This flexible document-oriented approach allows MongoDB to handle triples with varying structures efficiently.
- **Hive server:** Stores triples similarly to other databases, but in a distributed file system rather than a traditional database table. Each triple is represented as a row in a Hive table, with the table structured to include columns for the subject, predicate, and object components. This tabular format enables efficient querying and processing of triple data within the Hive ecosystem.

V. Testing

For testing our application, we have used the yago dataset, where the data is stored in subject predicate and object form. While testing our system, we have faced some difficulties in using hive, because as the dataset is large,hive is taking significant amount of time to perform update and merge operations. Apart from that we haven't faced any difficulties in testing our application.

So, to test our application whether it's working as it is supposed to be, first we performed query operations in all three servers and cross-checked if all three servers are giving consistent results. Next to check update operation, we performed update operation in one server giving subject, predicate and object values. after the server sent acknowledgement thorough socket programming, we again queried the same server with the same subject that was used in update operation to see whether changes are actually reflected in the database. And also checked the updates log database whether the update information got stored in it or not.

For testing merge operation, first we did some update operations in the server 1 and closed the connection with it and connected with the server 2. Now, in the server 2 we have performed the merge operation specifying the server 1 as the server with which merge operation should be performed. And checked if the server 2 has updated it's database making

changes performed by server 1 by iterating through the updates log table of server 1.

VI. Advanced Features

We have implemented socket programming using python for communication between client and server. This made the process of communication between client and server easy and simpler.And also we can close the connection with the current server at any time we wish and open connection with the other server.So, through this socket programming, shifting between servers is simple and user friendly.

Given our modular code implementation, incorporating a new server is a seamless process requiring minimal adjustments to the existing codebase. Specifically, in the merge function of the current servers, the only necessary modification entails adding an additional conditional block to accommodate the new server. This streamlined approach ensures that our system remains flexible and scalable, allowing for the effortless integration of new servers without the need for extensive code modifications or overhauls.

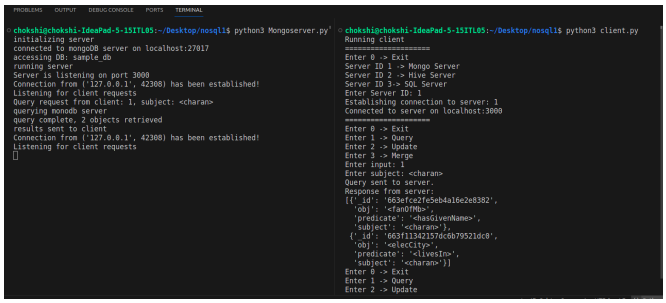
VII. User Interface

User can interact with the the system though command-line interface. First we make the client file and all server files to run in background.Now in the client program, user will be asked to which server to connect with, after getting reply from client, a connection will be established with the respective server and communication goes on.

So to connect with the hive server user is supposed to press 2, for mongo 1, similarly for postgres 3.Afer conecting, user is supposed to enter 1 for querying, 2 for updating and 3 for merging.Now, if at anytime user wants to close connection, he just needs to enter 0.for querying, user is required to give subject,for update he is required to provide subject,predicate and object values and for merge operation, user needs to enter the server id of the server with which he wants to perform merge operation.

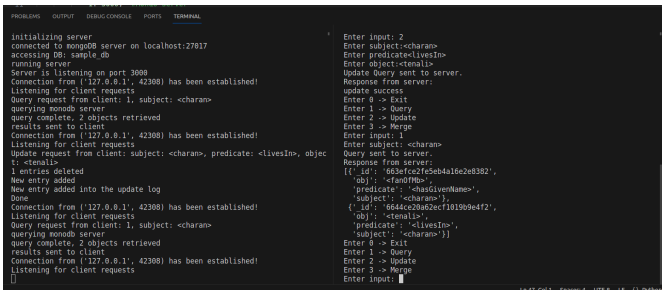
VIII. Evaluation & Results

First we tested query operation after connecting the client with mongo server. below is the result:

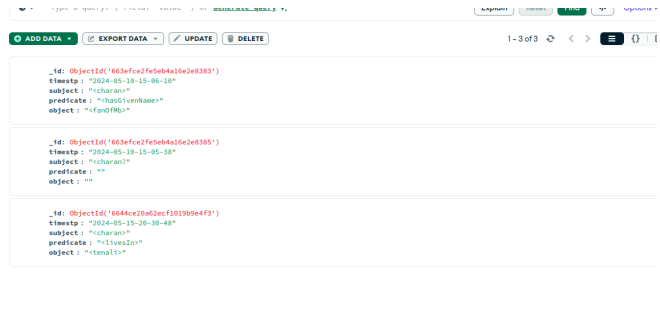


So, for the query with subject $< charan >$ it gave above rows. Now, we performed the update operation in the same server, changed the row with subject $< charan >$ and predicate $< livesIn >$ to the new object as $< tenali >$.

Below image depicts the reflected changes:

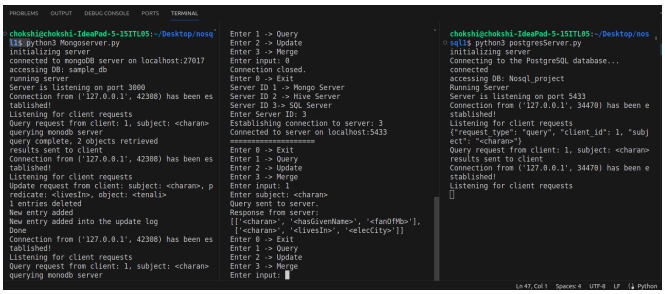


Below figure is the updates log of mongo server:



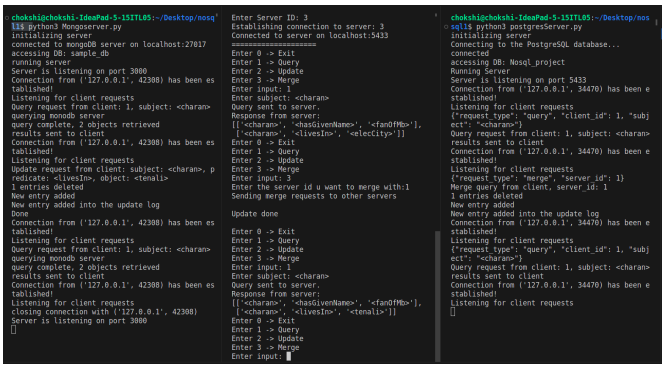
As you can see, a new update log with with values $< charan >, < livesIn >, < tenali >$ has been added.

Now, we closed the connection with mongoserver and connected to postgres server. Below is the query result with subject $< charan >$ in postgresserver:



So, if you see, the row with subject $< charan >$ and predicate $< livesIn >$ has still object value as $< elecCity >$.Now we performed merge operation in the postgres server with mongo server.

Below are the results after performing merge operation:



As you can see, after performing merge operation with

mongo server,changes made in mongo server are reflected in postgres server. object value of row with subject $< charan >$, predicate $< livesIn >$ has been changed from $< elecCity >$ to $< \ tenali \ >$. Thus, our system is functioning as it is supposed to be.

IX. Conclusion

In conclusion, our project successfully implemented a distributed NoSQL triple store prototype, showcasing the effective utilization of state-based objects and leveraging Python for seamless database interaction. Key achievements include the development of a modular system architecture capable of querying, updating, and merging triples across multiple servers, each employing distinct frameworks for data processing and storage. By utilizing socket programming, we facilitated efficient communication between clients and servers, enhancing the system's usability and flexibility. Moreover, the integration of advanced features such as dynamic server selection and user-friendly command-line interface further enhanced the project's functionality and usability.

Despite the successes, our project also encountered several challenges, particularly in testing and performance optimization. The large dataset used for testing posed scalability issues, especially with Hive, necessitating additional optimizations to improve efficiency. Additionally, ensuring data consistency and integrity during merge operations required careful handling of timestamps and updates logs, highlighting the importance of robust synchronization mechanisms in distributed systems.

Throughout the project, we gained valuable insights into distributed systems design, database management, and software development practices. The iterative nature of our development process allowed us to refine our implementation, address challenges, and incorporate feedback effectively. Moving forward, these experiences will serve as valuable lessons for tackling similar projects in the future, emphasizing the importance of scalability, fault tolerance, and usability in distributed system design.

Overall, our project demonstrates the feasibility and potential of distributed NoSQL triple stores in addressing complex data management challenges, offering scalable and resilient solutions for diverse application domains.